The Unreasonable Effectiveness of Recurrent Neural Networks

May 21, 2015

There's something magical about Recurrent Neural Networks (RNNs). I still remember when I trained my first recurrent network for Image Captioning. Within a few dozen minutes of training my first baby model (with rather arbitrarily-chosen hyperparameters) started to generate very nice looking descriptions of images that were on the edge of making sense. Sometimes the ratio of how simple your model is to the quality of the results you get out of it blows past your expectations, and this was one of those times. What made this result so shocking at the time was that the common wisdom was that RNNs were supposed to be difficult to train (with more experience I've in fact reached the opposite conclusion). Fast forward about a year: I'm training RNNs all the time and I've witnessed their power and robustness many times, and yet their magical outputs still find ways of amusing me. This post is about sharing some of that magic with you.

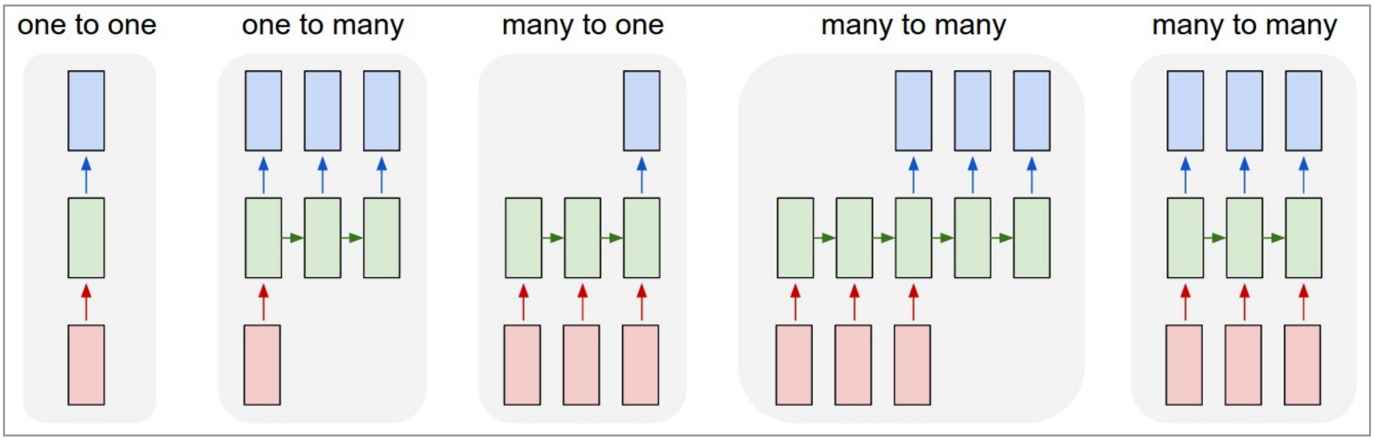
> *We'll train RNNs to generate text character by character and ponder the question "how is that even possible?"*

By the way, together with this post I am also releasing code on Github that allows you to train character-level language models based on multi-layer LSTMs. You give it a large chunk of text and it will learn to generate text like it one character at a time. You can also use it to reproduce my experiments below. But we're getting ahead of ourselves; What are RNNs anyway?

Recurrent Neural Networks

Sequences. Depending on your background you might be wondering: *What makes Recurrent Networks so special*? A glaring limitation of Vanilla Neural Networks (and also Convolutional Networks) is that their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) and produce a fixed-sized vector as output (e.g.

probabilities of different classes). Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model). The core reason that recurrent nets are more exciting is that they allow us to operate over *sequences* of vectors: Sequences in the input, the output, or in the most general case both. A few examples may make this more concrete:

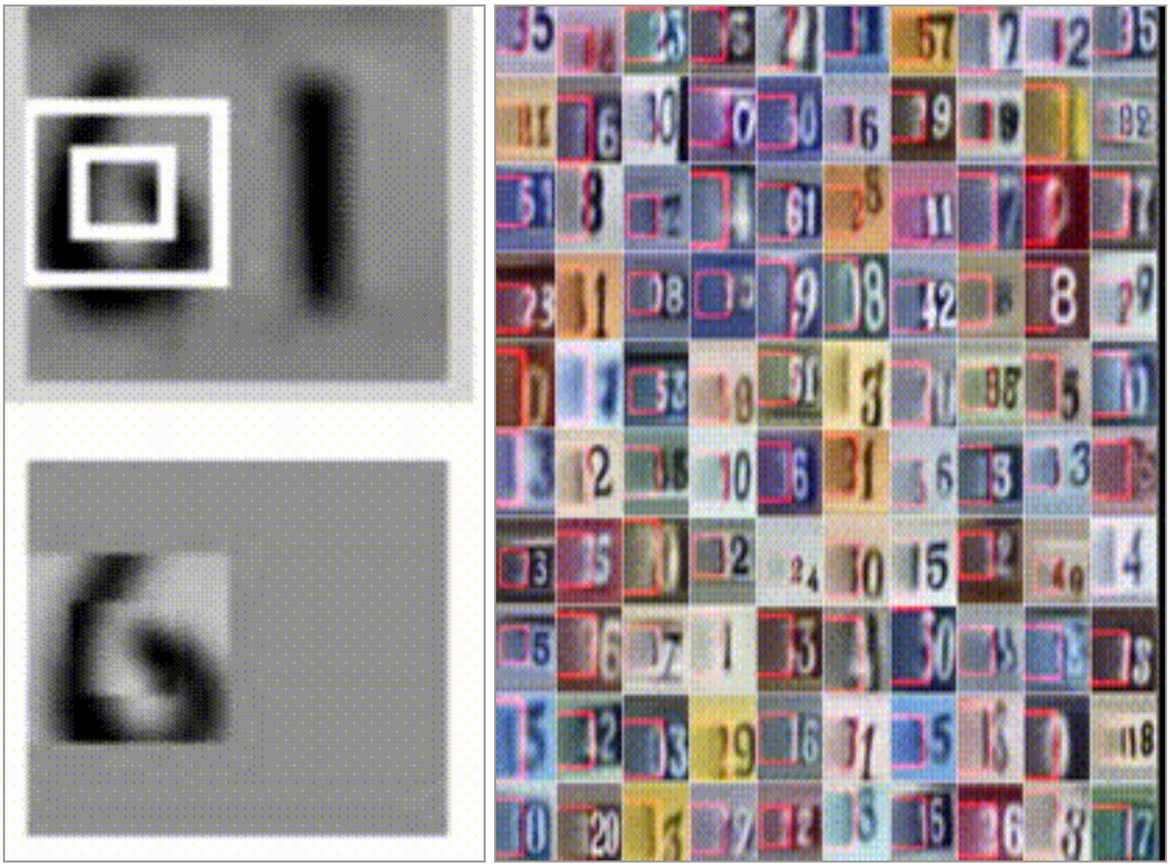


Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

As you might expect, the sequence regime of operation is much more powerful compared to fixed networks that are doomed from the get-go by a fixed number of computational steps, and hence also much more appealing for those of us who aspire to build more intelligent systems. Moreover, as we'll see in a bit, RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector. This can in programming terms be interpreted as running a fixed program with certain inputs and some internal variables. Viewed this way, RNNs essentially describe programs. In fact, it is known that RNNs are Turing-Complete in the sense that they can to simulate arbitrary programs (with proper weights). But similar to universal approximation theorems for neural nets you shouldn't read too much into this. In fact, forget I said anything.

> *If training vanilla neural nets is optimization over functions, training recurrent nets is optimization over programs.*

Sequential processing in absence of sequences. You might be thinking that having sequences as inputs or outputs could be relatively rare, but an important point to realize is that even if your inputs/outputs are fixed vectors, it is still possible to use this powerful formalism to *process* them in a sequential manner. For instance, the figure below shows results from two very nice papers from DeepMind. On the left, an algorithm learns a recurrent network policy that steers its attention around an image; In particular, it learns to read out house numbers from left to right (Ba et al.). On the right, a recurrent network *generates* images of digits by learning to sequentially add color to a canvas (Gregor et al.):



Left: RNN learns to read house numbers. Right: RNN learns to paint house numbers.

The takeaway is that even if your data is not in form of sequences, you can still formulate and train powerful models that learn to process it sequentially. You're learning stateful programs that process your fixed-sized data.

RNN computation. So how do these things work? At the core, RNNs have a deceptively simple API: They accept an input vector `x` and give you an output vector `y`. However, crucially this output vector's contents are influenced not only by the input you just fed in, but also on the entire history of inputs you've fed in in the past. Written as a class, the RNN's API consists of a single `step` function:

```
rnn = RNN()
y = rnn.step(x) # x is an input vector, y is the RNN's output vector
```

The RNN class has some internal state that it gets to update every time `step` is called. In the simplest case this state consists of a single *hidden* vector `h`. Here is an implementation of the step function in a Vanilla RNN:

```
class RNN:
  # ...
  def step(self, x):
    # update the hidden state
    self.h = np.tanh(np.dot(self.W_hh, self.h) + np.dot(self.W_xh, x))
    # compute the output vector
    y = np.dot(self.W_hy, self.h)
    return y
```

The above specifies the forward pass of a vanilla RNN. This RNN's parameters are the three matrices `W_hh, W_xh, W_hy`. The hidden state `self.h` is initialized with the zero vector. The `np.tanh` function implements a non-linearity that squashes the activations to the range `[-1, 1]`. Notice briefly how this works: There are two terms inside of the tanh: one is based on the previous hidden state and one is based on the current input. The two intermediates interact with addition, and then get squashed by the tanh into the new state vector.

We initialize the matrices of the RNN with random numbers and the bulk of work during training goes into finding the matrices that give rise to desirable behavior, as measured with some loss function that expresses your preference to what kinds of outputs `y` you'd like to see in response to your input sequences `x`.

Going deep. RNNs are neural networks and everything works monotonically better (if done right) if you put on your deep learning hat and start stacking models up like pancakes. For instance, we can form a 2-layer recurrent network as follows:

```
y1 = rnn1.step(x)
y = rnn2.step(y1)
```

In other words we have two separate RNNs: One RNN is receiving the input vectors and the second RNN is receiving the output of the first RNN as its input. Except neither of these RNNs know or care - it's all just vectors coming in and going out, and some gradients flowing through each module during backpropagation.

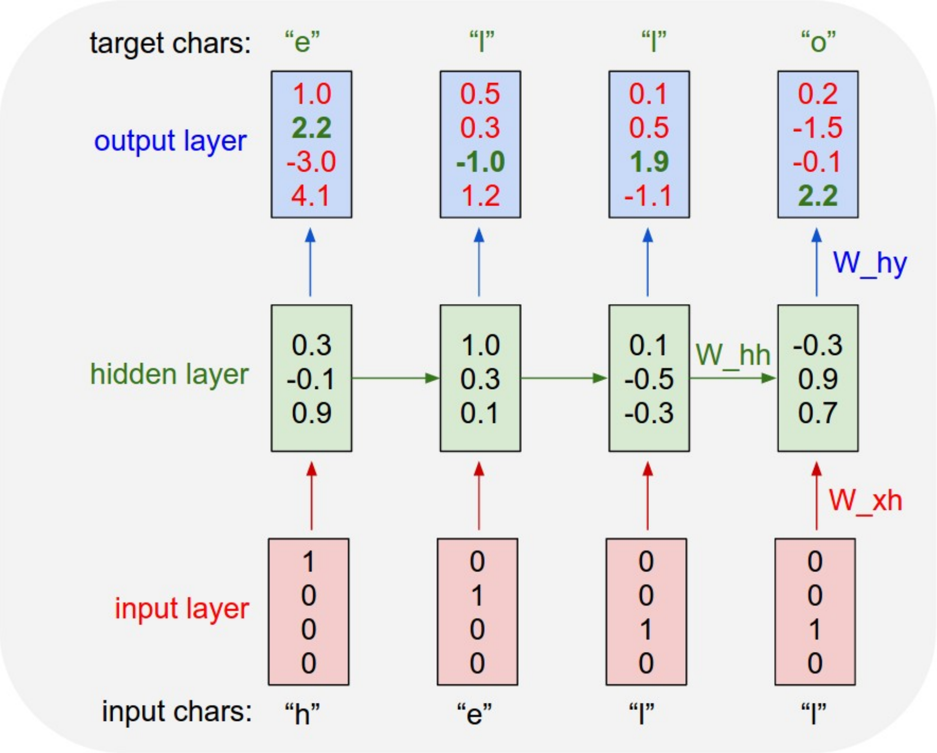
Getting fancy. I'd like to briefly mention that in practice most of us use a slightly different formulation than what I presented above called a *Long Short-Term Memory* (LSTM) network. The LSTM is a particular type of recurrent network that works slightly better in practice, owing to its more powerful update equation and some appealing backpropagation dynamics. I won't go into details, but everything I've said about RNNs stays exactly the same, except the mathematical form for computing the update (the line `self.h = ...`) gets a little more complicated. From here on I will use the terms "RNN/LSTM" interchangeably but all experiments in this post use an LSTM.

Character-Level Language Models

Okay, so we have an idea about what RNNs are, why they are super exciting, and how they work. We'll now ground this in a fun application: We'll train RNN character-level language models. That is, we'll give the RNN a huge chunk of text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters. This will then allow us to generate new text one character at a time.

As a working example, suppose we only had a vocabulary of four possible letters "helo", and wanted to train an RNN on the training sequence "hello". This training sequence is in fact a source of 4 separate training examples: 1. The probability of "e" should be likely given the context of "h", 2. "l" should be likely in the context of "he", 3. "l" should also be likely given the context of "hel", and finally 4. "o" should be likely given the context of "hell".

Concretely, we will encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary), and feed them into the RNN one at a time with the `step` function. We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence. Here's a diagram:



An example RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons). This diagram shows the activations in the forward pass when the RNN is fed the characters "hell" as input. The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o"); We want the green numbers to be high and red numbers to be low.

For example, we see that in the first time step when the RNN saw the character "h" it assigned confidence of 1.0 to the next letter being "h", 2.2 to letter "e", -3.0 to "l", and 4.1 to "o". Since in our training data the next correct character is "e", we would want to increase its confidence (green) and decrease the confidence of all other letters (red). The most common approach is to use a cross-entropy loss function here, which corresponds to placing a Softmax classifier on every output vector, with the correct class being the index of the next character in the sequence. Once the loss is backpropagated and the RNN's weights are updated, the correct next letters will have higher scores when faced with similar inputs.

Notice also that the first time the character "l" is input, the target is "l", but the second time the target is "o". The RNN therefore cannot rely on the input alone and must use its recurrent connection to keep track of the context to achieve this task.

At **test time**, we feed a character into the RNN and get a distribution over what characters are likely to come next. We sample from this distribution, and feed it right back in to get the next letter. Repeat this process and you're sampling text! Lets now train an RNN on different datasets and see what happens.

Fun with RNNs

All 5 example character models below were trained with the code I'm releasing on Github. The input in each case is a single file with some text, and we're training an RNN to predict the next character in the sequence.

Paul Graham generator

Lets first try a small dataset of English as a sanity check. My favorite fun dataset is the concatenation of Paul Graham's essays. The basic idea is that there's a lot of wisdom in these essays, but unfortunately Paul Graham is a relatively slow generator. Wouldn't it be great if we could sample startup wisdom on demand? That's where an RNN comes in.

Concatenating all pg essays over the last ~5 years we get approximately 1MB text file, or about 1 million characters (this is considered a very small dataset by the way). *Technical:* Lets train a 2-layer LSTM with 512 hidden nodes (approx. 3.5 million parameters), and with dropout of 0.5 after each layer. We'll train with batches of 100 examples and truncated backpropagation through time of length 100 characters. With these settings one batch on a TITAN Z GPU takes about 0.46 seconds (this can be cut in half with 50 character BPTT at negligible cost in performance). Without further ado, lets see a sample from the RNN:

"The surprised in investors weren't going to raise money. I'm not the company with the time there are all interesting quickly, don't have to get off the same programmers. There's a super-angel round fundraising, why do you can do. If you have a different physical investment are become in people who reduced in a startup with the way to argument the acquirer could see them just that you're also the founders will part of users' affords that and an alternation to the idea. [2] Don't work at first member to see the way kids will seem in advance of a bad successful startup. And if you have to act the big company too."

Okay, clearly the above is unfortunately not going to replace Paul Graham anytime soon, but remember that the RNN had to learn English completely from scratch and with a small dataset (including where you put commas, apostrophes and spaces). I also like that it learns to support its own arguments (e.g. [2], above). Sometimes it says something that offers a glimmer of insight, such as *"a company is a meeting to think to investors"*. Here's a link to 50K character sample if you'd like to see more.

Temperature. We can also play with the temperature of the Softmax during sampling. Decreasing the temperature from 1 to some lower number (e.g. 0.5) makes the RNN more confident, but also more conservative in its samples. Conversely, higher temperatures will give more diversity but at cost of more mistakes (e.g. spelling mistakes, etc). In particular, setting temperature very near zero will give the most likely thing that Paul Graham might say:

"is that they were all the same thing that was a startup is that they were all the same thing that was a startup is that they were all the same thing that was a startup is that they were all the same"

looks like we've reached an infinite loop about startups.

Shakespeare

It looks like we can learn to spell English words. But how about if there is more structure and style in the data? To examine this I downloaded all the works of Shakespeare and concatenated them into a single (4.4MB) file. We can now afford to train a larger network, in this case lets try a 3-layer RNN with 512 hidden nodes on each layer. After we train the network for a few hours we obtain samples such as:

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
And who is but a chain and subjects of his death,
I should not sleep.

Second Senator:
They are away this miseries, produced upon my soul,
Breaking and strongly should be buried, when I perish
The earth and thoughts of many states.

DUKE VINCENTIO:
Well, your wit is in the care of side and that.

Second Lord:
They would be ruled after this chamber, and
my fair nues begun out of the fact, to be conveyed,
Whose noble souls I'll have the heart of the wars.
```

```
Clown:
Come, sir, I will make did behold your worship.

VIOLA:
I'll drink it.
```

Remember, all the RNN knows are charaters, so in particular it samples both speaker's names and the contents. Sometimes we also get relatively extented monologue passages, such as:

```
VIOLA:
Why, Salisbury must find his flesh and thought
That which I am not aps, not a man and in fire,
To show the reining of the raven and the wars
To grace my hand reproach within, and not a fair are hand,
That Caesar and my goodly father's world;
When I was heaven of presence and our fleets,
We spare with hours, but cut thy council I am great,
Murdered and by thy master's ready there
My power to give thee but so much as hell:
Some service in the noble bondman here,
Would show him to her wine.

KING LEAR:
O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.
```

I can barely recognize these samples from actual Shakespeare :) If you like Shakespeare, you might appreciate this 100,000 character sample. Of course, you can also generate an infinite amount of your own samples at different temperatures with the provided code.

Wikipedia

We saw that the LSTM can learn to spell words and copy general syntactic structures.

Lets further increase the difficulty and train on structured markdown. In particular, lets take the Hutter Prize 100MB dataset of raw Wikipedia and train an LSTM. Following Graves et al., I used the first 96MB for training, the rest for validation and ran a few models overnight. Alex reports performance of 1.67 Bits Per Character (BPC) with a 7-layer LSTM of 700 nodes. My best model ended up being a 3-layer LSTM with 700 nodes, achieving 1.57 BPC after 7 epochs of training. I'm not exactly sure what accounts for my better performance. Regardless, we can now sample Wikipedia articles! Below are a few fun excerpts. First, some basic markdown output:

```
Naturalism and decision for the majority of Arab countries' capitalide
by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt
with Guangzham's sovereignty. His generals were the powerful ruler of
in the [[Protestant Immineners]], which could be said to be directly i
Communication, which followed a ceremony and set inspired prison, trai
emperor travelled back to [[Antioch, Perth, October 25|21]] to note, t
of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoa
in western [[Scotland]], near Italy to the conquest of India with the 
Copyright was the succession of independence in the slop of Syrian inf
was a famous German movement based on a more popular servicious, non-c
and sexual power post. Many governments recognize the military housing
[[Civil Liberalization and Infantry Resolution 265 National Party in H
that is sympathetic to be to the [[Punjab Resolution]]
(PJS)[http://www.humah.yahoo.com/guardian.
cfm/7754800786d17551963s89.htm Official economics Adjoint for the Nazi
was swear to advance to the resources for those Socialism's rule,
was starting to signing a major tripad of aid exile.]]
```

In case you were wondering, the yahoo url above doesn't actually exist, the model just hallucinated it. Also, note that the model learns to open and close the parenthesis correctly. There's also quite a lot of structured markdown that the model learns, for example sometimes it creates headings, lists, etc.:

```
{ { cite journal | id=Cerling Nonforest Department|format=Newlymeslate
''www.e-complete''.

'''See also''': [[List of ethical consent processing]]

== See also ==
*[[Iender dome of the ED]]
```

```
*[[Anti-autism]]

===[[Religion|Religion]]===
*[[French Writings]]
*[[Maria]]
*[[Revelation]]
*[[Mount Agamul]]

== External links==
* [http://www.biblegateway.nih.gov/entrepre/ Website of the World Fest

==External links==
* [http://www.romanology.com/ Constitution of the Netherlands and Hisp
```

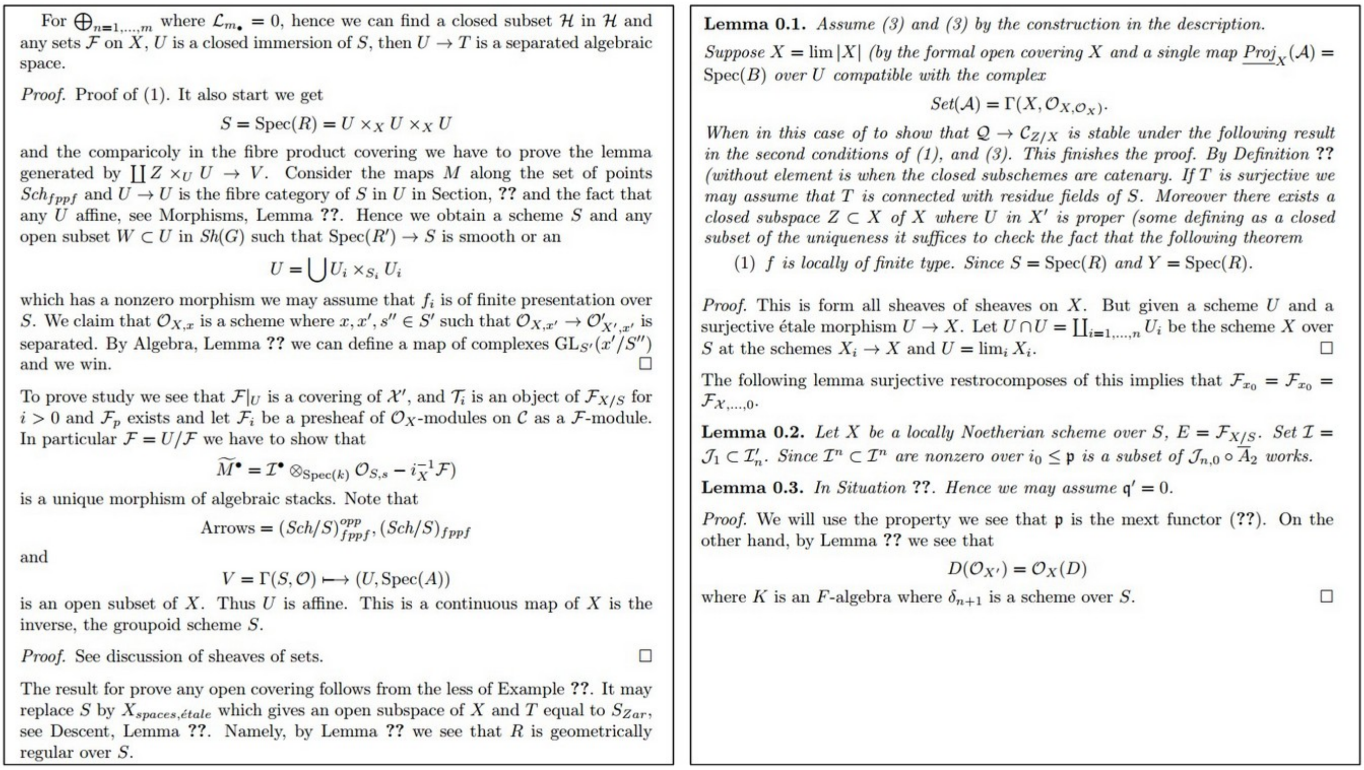
Sometimes the model snaps into a mode of generating random but valid XML:

```
<page>
  <title>Antichrist</title>
  <id>865</id>
  <revision>
    <id>15900676</id>
    <timestamp>2002-08-03T18:14:12Z</timestamp>
    <contributor>
      <username>Paris</username>
      <id>23</id>
    </contributor>
    <minor />
    <comment>Automated conversion</comment>
    <text xml:space="preserve">#REDIRECT [[Christianity]]</text>
  </revision>
</page>
```

The model completely makes up the timestamp, id, and so on. Also, note that it closes the correct tags appropriately and in the correct nested order. Here are 100,000 characters of sampled wikipedia if you're interested to see more.

Algebraic Geometry (Latex)

The results above suggest that the model is actually quite good at learning complex syntactic structures. Impressed by these results, my labmate (Justin Johnson) and I decided to push even further into structured territories and got a hold of this book on algebraic stacks/geometry. We downloaded the raw Latex source file (a 16MB file) and trained a multilayer LSTM. Amazingly, the resulting sampled Latex *almost* compiles. We had to step in and fix a few issues manually but then you get plausible looking math, it's quite astonishing:



For $\bigoplus_{n=1,\dots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the comparicoly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, **??** and the fact that any U affine, see Morphisms, Lemma **??**. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma **??** we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. □

The result for prove any open covering follows from the less of Example **??**. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma **??**. Namely, by Lemma **??** we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ *(by the formal open covering* X *and a single map* $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ *over* U *compatible with the complex*

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ *is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition* **??** *(without element is when the closed subschemes are catenary. If* T *is surjective we may assume that* T *is connected with residue fields of* S*. Moreover there exists a closed subspace* $Z \subset X$ *of* X *where* U *in* X' *is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem*

(1) f *is locally of finite type. Since* $S = \mathrm{Spec}(R)$ *and* $Y = \mathrm{Spec}(R)$.

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\dots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. □

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{\mathcal{X},\dots,0}$.

Lemma 0.2. *Let* X *be a locally Noetherian scheme over* S, $E = \mathcal{F}_{X/S}$*. Set* $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$*. Since* $\mathcal{I}^n \subset \mathcal{I}^n$ *are nonzero over* $i_0 \leq \mathfrak{p}$ *is a subset of* $\mathcal{J}_{n,0} \circ \overline{A}_2$ *works.*

Lemma 0.3. *In Situation* **??***. Hence we may assume* $\mathfrak{q}' = 0$.

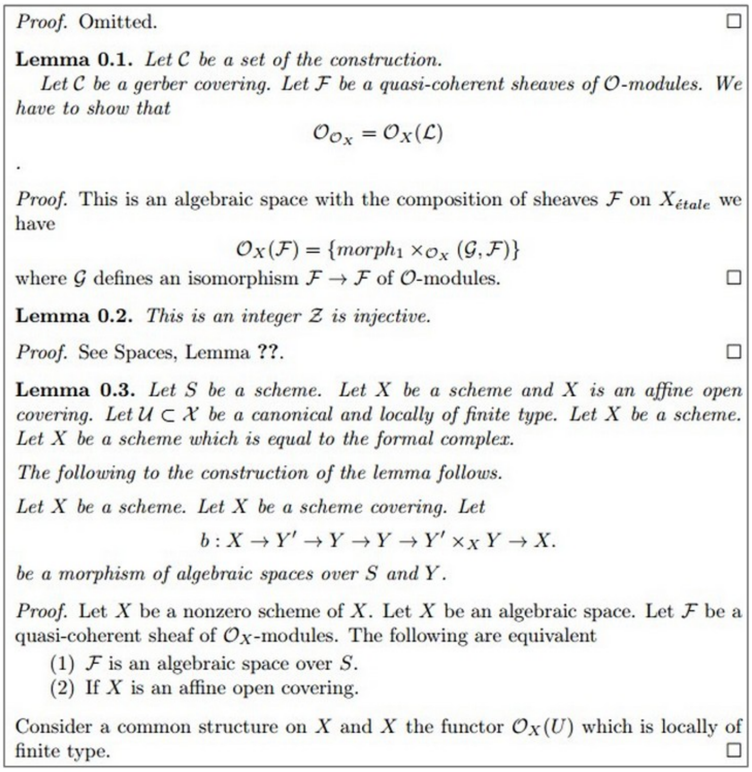
Proof. We will use the property we see that $\mathfrak{p}$ is the mext functor (**??**). On the other hand, by Lemma **??** we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F-algebra where δ_{n+1} is a scheme over S. □

Sampled (fake) algebraic geometry. Here's the actual pdf.

Here's another sample:



Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*

Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma **??**. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

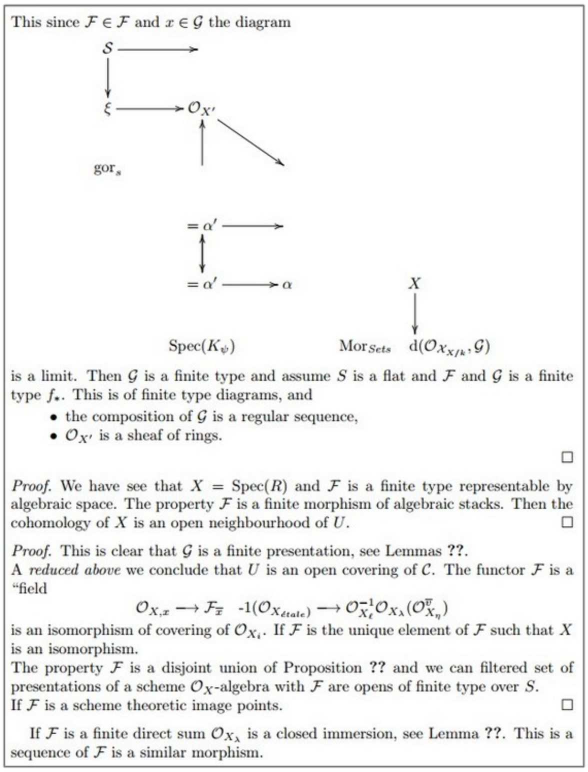
$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □



This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{tikzcd}
\mathcal{S} \arrow[r] \arrow[d] & {} & & & \\
\xi \arrow[r] & \mathcal{O}_{X'} \arrow[dr] & & & \\
\mathrm{gor}_s & {} \arrow[u] & {} & & \\
& =\alpha' \arrow[r] \arrow[d, leftrightarrow] & {} & & \\
& =\alpha' \arrow[r] & \alpha & & X \arrow[d] \\
& \mathrm{Spec}(K_\psi) & & \mathrm{Mor}_{Sets} & \mathrm{d}(\mathcal{O}_{\mathcal{X}_{X/k}}, \mathcal{G})
\end{tikzcd}$$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \mathrm{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas **??**.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \ \ \text{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_\ell}^{-1}\mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\overline{v}})$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition **??** and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma **??**. This is a sequence of $\mathcal{F}$ is a similar morphism.

More hallucinated algebraic geometry. Nice try on the diagram (right).

As you can see above, sometimes the model tries to generate latex diagrams, but clearly it hasn't really figured them out. I also like the part where it chooses to skip a proof (*"Proof omitted."*, top left). Of course, keep in mind that latex has a relatively difficult structured syntactic format that I haven't even fully mastered myself. For instance, here is a raw sample from the model (unedited):

```
\begin{proof}
We may assume that $\mathcal{I}$ is an abelian sheaf on $\mathcal{C}$.
\item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$
is an injective and let $\mathfrak q$ be an abelian sheaf on $X$.
Let $\mathcal{F}$ be a fibered complex. Let $\mathcal{F}$ be a catego
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
Let $\mathcal{F}$ be an abelian quasi-coherent sheaf on $\mathcal{C}$.
Let $\mathcal{F}$ be a coherent $\mathcal{O}_X$-module. Then
$\mathcal{F}$ is an abelian catenary over $\mathcal{C}$.
\item The following are equivalent
\begin{enumerate}
\item $\mathcal{F}$ is an $\mathcal{O}_X$-module.
\end{lemma}
```

This sample from a relatively decent model illustrates a few common mistakes. For example, the model opens a `\begin{proof}` environment but then ends it with a `\end{lemma}`. This is an example of a problem we'd have to fix manually, and is likely due to the fact that the dependency is too long-term: By the time the model is done with the proof it has forgotten whether it was doing a proof or a lemma. Similarly, it opens an `\begin{enumerate}` but then forgets to close it. We observed that these became less common with larger/better models, but nonetheless, these are the kinds of mistakes that come up.

Linux Source Code

I wanted to push structured data to its limit, so for the final challenge I decided to use code. In particular, I took all the source and header files found in the Linux repo on Github, concatenated all of them in a single giant file (474MB of C code) (I was originally going to train only on the kernel but that by itself is only ~16MB). Then I trained several as-large-as-fits-on-my-GPU 3-layer LSTMs over a period of a few days. These models have about 10 million parameters, which is still on the lower end for RNN models. The results are superfun:

```
/*
 * Increment the size file of the new incorrect UI_FILTER group infor
 * of the size generatively.
 */
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHIN
  return segtable;
}
```

The code looks really quite great overall. Of course, I don't think it compiles but when you scroll through the generate code it feels very much like a giant C code base. Notice that the RNN peppers its code with comments here and there at random. It is also very good at making very few syntactic errors. For example, it uses strings properly, pointer notation, etc. It also opens and closes brackets `{[` correctly and learns to indent its code very well. A common error is that it can't keep track of variable names: It often uses undefined variables (e.g. `rw` above), declares variables it never uses (e.g. `int error`), or returns non-existing variables. Lets see a few more examples. Here's another snippet that shows a wider array of operations that the RNN learns:

```
/*
 * If this error is set, we will need anything right after that BSD.
 */
static void action_new_function(struct s_stat_info *wb)
{
  unsigned long flags;
  int lel_idx_bit = e->edd, *sys & ~((unsigned long) *FIRST_COMPAT);
  buf[0] = 0xFFFFFFFF & (bit << 4);
  min(inc, slist->bytes);
  printk(KERN_WARNING "Memory allocated %02x/%02x, "
    "original MLL instead\n"),
    min(min(multi_run - s->len, max) * num_data_in),
    frame_pos, sz + first_seg);
  div_u64_w(val, inb_p);
  spin_unlock(&disk->queue_lock);
  mutex_unlock(&s->sock->mutex);
  mutex_unlock(&func->mutex);
  return disassemble(info->pending_bh);
}

static void num_serial_settings(struct tty_struct *tty)
{
  if (tty == tty)
    disable_single_st_p(dev);
  pci_disable_spool(port);
  return 0;
}

static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

Notice that in the second function the model compares `tty == tty`, which is vacuously true. On the other hand, at least the variable `tty` exists in the scope this time! In the last function, notice that the code does not return anything, which happens to be correct since the function signature is `void`. However, the first two functions were also declared `void` and did return values. This is again a form of a common mistake due to long-term interactions.

Sometimes the model decides that it's time to sample a new file. This is usually a very amusing part: The model first recites the GNU license character by character, samples a few includes, generates some macros and then dives into the code:

```
/*
 *  Copyright (c) 2006-2010, Intel Mobile Communications.  All rights
 *
 *   This program is free software; you can redistribute it and/or mo
 * under the terms of the GNU General Public License version 2 as publ
 * the Free Software Foundation.
 *
 *        This program is distributed in the hope that it will be use
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *
 *  GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public Licens
 *    along with this program; if not, write to the Free Software Fou
 *  Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>

#define REG_PG    vesa_slot_addr_pack
#define PFM_NOCOMP  AFSR(0, load)
#define STACK_DDR(type)     (func)
```

```
#define SWAP_ALLOCATE(nr)     (e)
#define emulate_sigs()  arch_get_unaligned_child()
#define access_rw(TST)  asm volatile("movd %%esp, %0, %3" : : "r" (0))
  if (__type & DO_READ)

static void stat_PC_SEC __read_mostly offsetof(struct seq_argsqueue, \
          pC>[1]);

static void
os_prefix(unsigned long sys)
{
#ifdef CONFIG_PREEMPT
  PUT_PARAM_RAID(2, sel) = get_state_state();
  set_pid_sum((unsigned long)state, current_state_str(),
           (unsigned long)-1->lr_full; low;
}
```

There are too many fun parts to cover- I could probably write an entire blog post on just this part. I'll cut it short for now, but here is 1MB of sampled Linux code for your viewing pleasure.

Understanding what's going on

We saw that the results at the end of training can be impressive, but how does any of this work? Lets run two quick experiments to briefly peek under the hood.

The evolution of samples while training

First, it's fun to look at how the sampled text evolves while the model trains. For example, I trained an LSTM of Leo Tolstoy's War and Peace and then generated samples every 100 iterations of training. At iteration 100 the model samples random jumbles:

```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranby
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

However, notice that at least it is starting to get an idea about words separated by spaces. Except sometimes it inserts two spaces. It also doesn't know that comma is amost always followed by a space. At 300 iterations we see that the model starts to get an idea about quotes and periods:

```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseter
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

The words are now also separated with spaces and the model starts to get the idea about periods at the end of a sentence. At iteration 500:

```
we counter. He stutn co des. His stanted out one ofler that concossion
to gearang reay Jotrets and with fre colt otf paitt thin wall. Which
```

the model has now learned to spell the shortest and most common words such as "we", "He", "His", "Which", "and", etc. At iteration 700 we're starting to see more and more English-like text emerge:

```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say
how, and Gogition is so overelical and ofter.
```

At iteration 1200 we're now seeing use of quotations and question/exclamation marks. Longer words have now been learned as well:

```
"Kite vouch!" he repeated by her
door. "But I would be done and quarts, feeling, then, son is people..
```

Until at last we start to get properly spelled words, quotations, names, and so on by about iteration 2000:

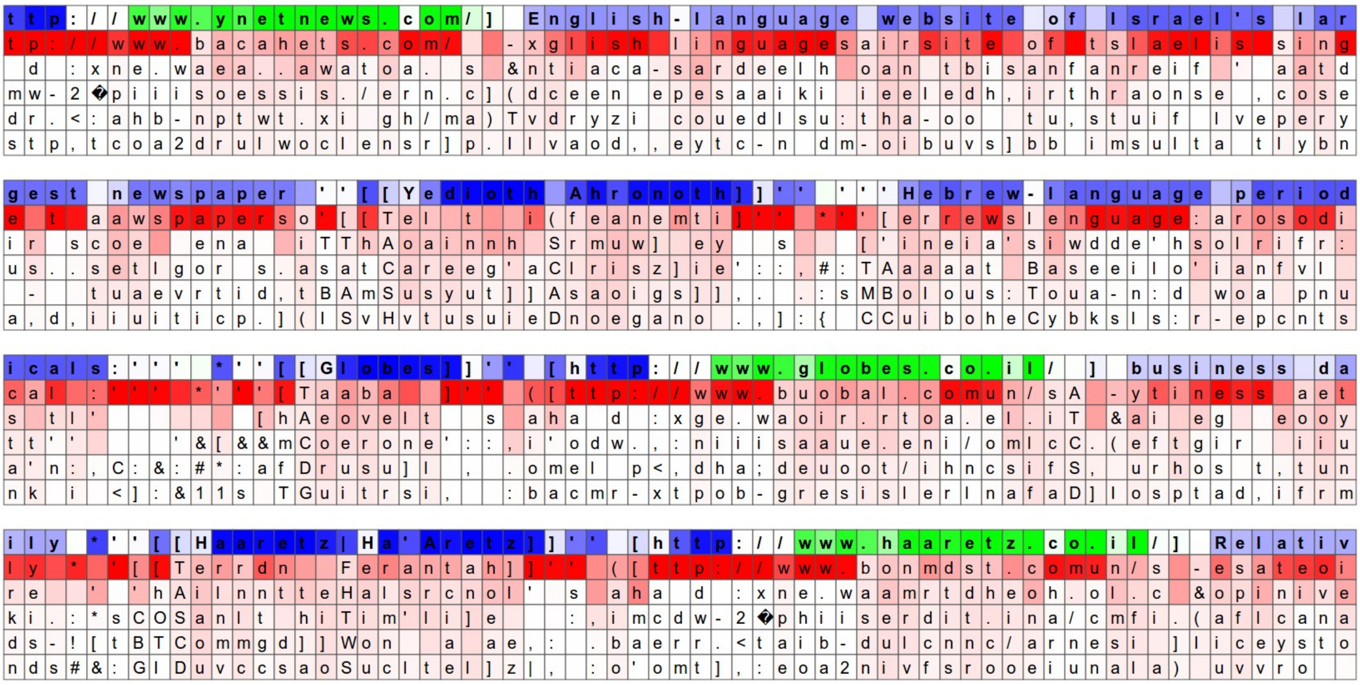
```
"Why do what that day," replied Natasha, and wishing to himself the fa
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-la
```

The picture that emerges is that the model first discovers the general word-space structure and then rapidly starts to learn the words; First starting with the short words and then eventually the longer ones. Topics and themes that span multiple words (and in general longer-term dependencies) start to emerge only much later.

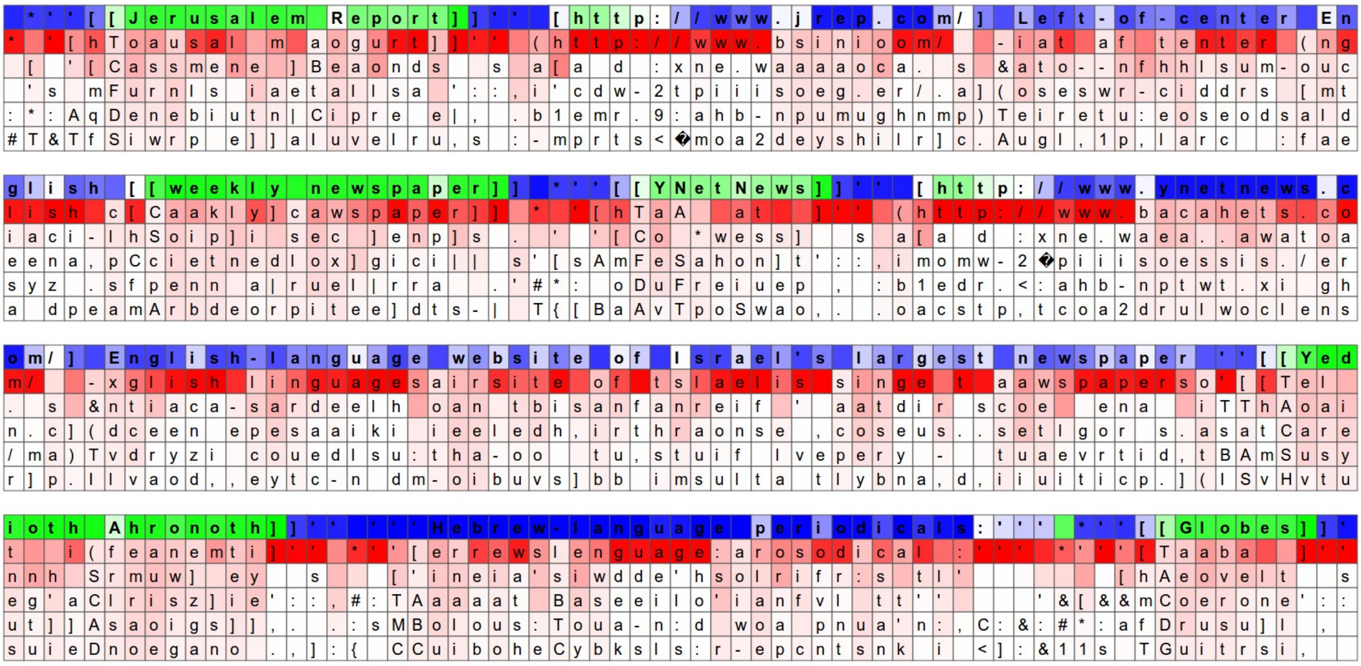
Visualizing the predictions and the "neuron" firings in the RNN

Another fun visualization is to look at the predicted distributions over characters. In the visualizations below we feed a Wikipedia RNN model character data from the validation set (shown along the blue/green rows) and under every character we visualize (in red) the top 5 guesses that the model assigns for the next character. The guesses are colored by their probability (so dark red = judged as very likely, white = not very likely). For example, notice that there are stretches of characters where the model is extremely confident about the next letter (e.g., the model is very confident about characters during the *http://www.* sequence).

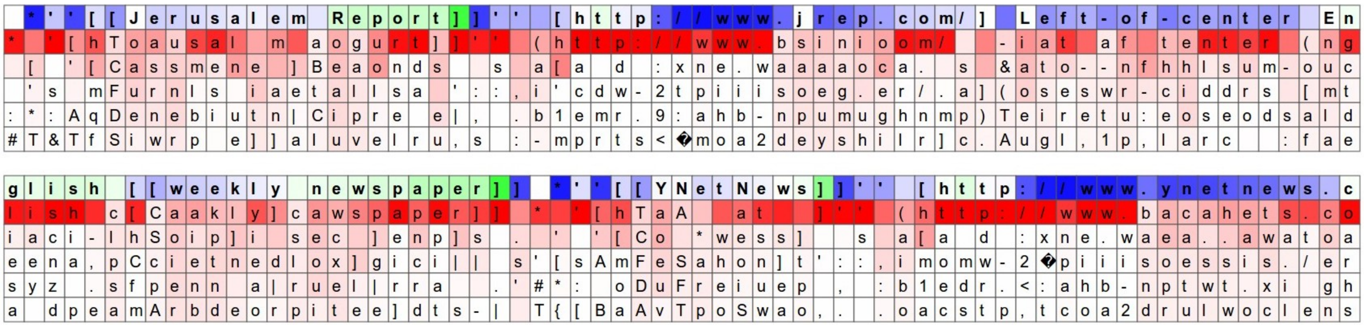
The input character sequence (blue/green) is colored based on the *firing* of a randomly chosen neuron in the hidden representation of the RNN. Think about it as green = very excited and blue = not very excited (for those familiar with details of LSTMs, these are values between [-1,1] in the hidden state vector, which is just the gated and tanh'd LSTM cell state). Intuitively, this is visualizing the firing rate of some neuron in the "brain" of the RNN while it reads the input sequence. Different neurons might be looking for different patterns; Below we'll look at 4 different ones that I found and thought were interesting or interpretable (many also aren't):



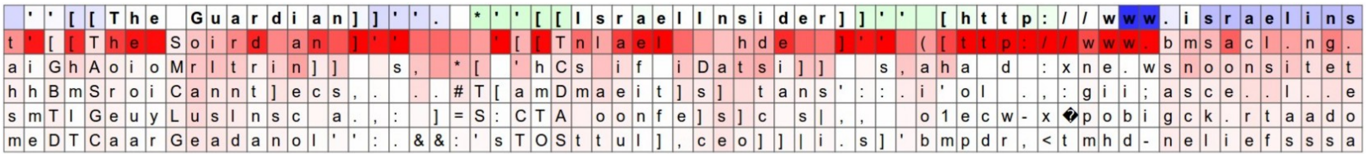
The neuron highlighted in this image seems to get very excited about URLs and turns off outside of the URLs. The LSTM is likely using this neuron to remember if it is inside a URL or not.



The highlighted neuron here gets very excited when the RNN is inside the [[ ]] markdown environment and turns off outside of it. Interestingly, the neuron can't turn on right after it sees the character "[", it must wait for the second "[" and then activate. This task of counting whether the model has seen one or two "[" is likely done with a different neuron.



Here we see a neuron that varies seemingly linearly across the [[ ]] environment. In other words its activation is giving the RNN a time-aligned coordinate system across the [[ ]] scope. The RNN can use this information to make different characters more or less likely depending on how early/late it is in the [[ ]] scope (perhaps?).



Here is another neuron that has very local behavior: it is relatively silent but sharply turns off right after the first "w" in the "www" sequence. The RNN might be using this neuron to count up how far in the "www" sequence it is, so that it can know whether it should emit another "w", or if it should start the URL.

Of course, a lot of these conclusions are slightly hand-wavy as the hidden state of the RNN is a huge, high-dimensional and largely distributed representation.

Source Code

I hope I've convinced you that training character-level language models is a very fun exercise. You can train your own models using the char-rnn code I released on Github (under MIT license). It takes one large text file and trains a character-level model that you can then sample from. Also, it helps if you have a GPU or otherwise training on CPU will be about a factor of 10x slower. In any case, if you end up training on some data and getting fun results let me know!

Brief digression. The code is written in Torch 7, which has recently become my favorite deep learning framework. I've only started working with Torch/LUA over the last few months and it hasn't been easy (I spent a good amount of time digging through the raw Torch code on Github and asking questions on their *gitter* to get things done), but once you get a hang of things it offers a lot of flexibility and speed. I've also worked with Caffe and Theano in the past and I believe Torch, while not perfect, gets its levels of abstraction and philosophy right better than others. In my view the desirable features of an effective framework are:

1. CPU/GPU transparent Tensor library with a lot of functionality (slicing, array/matrix operations, etc.)
2. An entirely separate code base in a scripting language (ideally Python) that operates over Tensors and implements all Deep Learning stuff (forward/backward, computation graphs, etc)
3. It should be possible to easily share pretrained models (Caffe does this well, others don't), and crucially
4. NO compilation step (or at least not as currently done in Theano). The trend in Deep Learning is towards larger, more complex networks that are are time-unrolled in complex graphs. It is critical that these do not compile for a long time or development time greatly suffers. Second, by compiling one gives up interpretability and the ability to log/debug effectively.

Further Reading

Before the end of the post I also wanted to position RNNs in a wider context and provide a sketch of the current research directions. RNNs have recently generated a significant amount of buzz and excitement in the field of Deep Learning. Similar to Convolutional Networks they have been around for decades but their full potential has only recently started to get widely recognized, in large part due to our growing computational resources. Here's a brief sketch of a few recent developments (definitely not complete list, and a lot of this work draws from research back to 1990s, see related work sections):

In the domain of **NLP/Speech**, RNNs transcribe speech to text, perform machine translation, generate handwritten text, and of course, they have been used as powerful language models (Sutskever et al.) (Graves) (Mikolov et al.) (both on the level of characters and words). Currently it seems that word-level models work better than character-level models, but this is surely a temporary thing.

Computer Vision. RNNs are also quickly becoming pervasive in Computer Vision. For example, we're seeing RNNs in frame-level video classification, image captioning (also including my own work and many others), video captioning and very recently visual question answering. My personal favorite RNNs in Computer Vision paper is Recurrent Models of Visual Attention, both due to its high-level direction (sequential processing of images with glances) and the low-level modeling (REINFORCE learning rule that is a special case of policy gradient methods in Reinforcement Learning, which allows one to train models that perform non-differentiable computation (taking glances around the image in this case)). I'm confident that this type of hybrid model that consists of a blend of CNN for raw perception coupled with an RNN glance policy on top will become pervasive in perception, especially for more complex tasks that go beyond classifying some objects in plain view.

Inductive Reasoning, Memories and Attention. Another extremely exciting direction of research is oriented towards addressing the limitations of vanilla recurrent networks. One problem is that RNNs are not inductive: They memorize sequences extremely well, but they don't necessarily always show convincing signs of generalizing in the *correct* way (I'll provide pointers in a bit that make this more concrete). A second issue is they unnecessarily couple their representation size to the amount of computation per step. For instance, if you double the size of the hidden state vector you'd quadruple the amount of FLOPS at each step due to the matrix multiplication. Ideally, we'd like to maintain a huge representation/memory (e.g. containing all of Wikipedia or many intermediate state variables), while maintaining the ability to keep computation per time step fixed.

The first convincing example of moving towards these directions was developed in DeepMind's Neural Turing Machines paper. This paper sketched a path towards models that can perform read/write operations between large, external memory arrays and a smaller set of memory registers (think of these as our working memory) where the computation happens. Crucially, the NTM paper also featured very interesting memory addressing mechanisms that were implemented with a (soft, and fully-differentiable) attention model. The concept of **soft attention** has turned out to be a powerful modeling features and was also featured in Neural Machine Translation by Jointly Learning to Align and Translate for Machine Translation and Memory Networks for (toy) Question Answering. In fact, I'd go as far as to say that

> *The concept of **attention** is the most interesting recent architectural innovation in neural networks.*

Now, I don't want to dive into too many details but a soft attention scheme for memory addressing is convenient because it keeps the model fully-differentiable, but unfortunately one sacrifices efficiency because everything that can be attended to is attended to (but softly). This has motivated multiple authors to swap soft attention models for **hard attention** where one samples a particular chunk of memory to attend to (e.g. a read/write action for some memory cell instead of reading/writing from all cells to some degree). This model is significantly more philosophically appealing, scalable and efficient, but unfortunately it is also non-differentiable. This then calls for use of techniques from the Reinforcement Learning literature (e.g. REINFORCE) where people are perfectly used to the concept of non-differentiable interactions. This is very much ongoing work but these hard attention models have been explored, for example, in Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets, Reinforcement Learning Neural Turing Machines, and Show Attend and Tell.

People. If you'd like to read up on RNNs I recommend theses from Alex Graves, Ilya Sutskever and Tomas Mikolov. For more about REINFORCE and more generally Reinforcement Learning and policy gradient methods (which REINFORCE is a special case of) David Silver's class, or one of Pieter Abbeel's classes.

Code. If you'd like to play with training RNNs I hear good things about keras or passage for Theano, the code released with this post for Torch, or this gist for raw numpy code I wrote a while ago that implements an efficient, batched LSTM forward and backward pass. You can also have a look at my numpy-based NeuralTalk which uses an RNN/LSTM to caption images, or maybe this Caffe implementation by Jeff Donahue.

Conclusion

We've learned about RNNs, how they work, why they have become a big deal, we've trained an RNN character-level language model on several fun datasets, and we've seen where RNNs are going. You can confidently expect a large amount of innovation in the space of RNNs, and I believe they will become a pervasive and critical component to intelligent systems.

Lastly, to add some **meta** to this post, I trained an RNN on the source file of this blog post. Unfortunately, at about 46K characters I haven't written enough data to properly feed the RNN, but the returned sample (generated with low temperature to get a more typical sample) is:

```
I've the RNN with and works, but the computed with program of the
RNN with and the computed of the RNN with with and the code
```

Yes, the post was about RNN and how well it works, so clearly this works :). See you next time!

EDIT (extra links): HN discussion, Reddit discussion.

45 Comments Andrej's Blog 1 Login ▾

♥ Recommend 25 Share Sort by Best ▾

Join the discussion…

RalfD · 14 hours ago

Maybe it would be fun to feed in musical notation and then hear the output? The Fake-Shakespeare and fake-programming produced here is impressive (even though nonsensical), but I wonder what would Beethoven, Bach or Mozart sound like? Chaotic or actually melodic?

13 ˄ | ˅ · Reply · Share ›

Adrià Garriga → RalfD · 11 hours ago

Try training it on some Lilypond or ABC files. Seeing what the C file looked like I say they will sound chaotic, because the structure of those files do not tie the simultaneous-sounding notes together.

However, if you restructure them to have measures played at the same time together in the source, the result might be better.

This remains as a project to do another day :)

2 ˄ | ˅ · Reply · Share ›

yankov → RalfD · 5 hours ago

I am wondering also what kind of output you get if you feed it visual data, for example paintings of Van Gogh. It will be certainly a mess, but will there be an interesting structure?

˄ | ˅ · Reply · Share ›

Mihail Sirotenko · 13 hours ago

Somebody has to combine LSTM learning to code with reinforcement learning, where reward would depend on the number of compiler errors and warnings for the resulting code.

3 ˄ | ˅ · Reply · Share ›

wtpayne → Mihail Sirotenko · 6 hours ago

And unit-test failures ...

Also, the training shouldn't be on the final source itself, but on the sequence of commits, so you can take a piece of source code and try to predict what the next commit will be ....

2 ⌃ | ⌄ • Reply • Share ›

Chris B • a day ago

Awesome as always. I'm really curious if it's possible to do a deeper analysis of the hidden state. Looking at individual neurons seems like it will run into trouble since most of the state information is probably distributed (right?). Some possibilities might be:
1) Looking at an embedding of all the RNN states - presumably the temporal sequence will be a slow walk through this space, though there might be jumps when important events happen (like punctuation).
2) Doing linear regression from hidden states to state properties - e.g. how well can you predict if you're inside parentheses, or the length of the current sentence, or what part of a URL you're currently in?
3) Somehow separating out the different temporal frequencies in the state. It seems like there are some pieces of information that need to change very quickly (e.g. to predict the next letter) and others that need to be stable over long time periods. You could do this for individual units by looking at something like the width of the autocorrelation, or you could apply this to distributed components from something like PCA/ICA.
4) Is it possible to analyze the learned weight matrices, e.g. W_hh in any way? Does this W_hh end up being full rank? Does it have meaningful eigenvectors in any sense?

see more

2 ⌃ | ⌄ • Reply • Share ›

a2480f25 • 6 minutes ago

Problem : Catastrophic forgetting (interference) - the tendency of a artificial neural network to completely and abruptly forget previously learned information upon learning new information.

Proposed solution : Neural Modularity Helps Organisms Evolve to Learn New Skills without Forgetting Old Skills

[...] To evolve modular networks, we add another natural phenomenon: costs for neural connections. In nature, there are many costs associated with neural connections (e.g. building them, maintaining them, and housing them) [26–28] and it was recently demonstrated that incorporating a cost for such connections encourages the evolution of modularity in networks [23]. [...]

http://journals.plos.org/plosc...

⌃ | ⌄ • Reply • Share ›

oggy · 5 hours ago

In all the examples on the page, the RNN is first trained and then used to generate the text. Is there a way to use RNNs for something more reactive? Say, something like a chatbot. Can one train an RNN to mimic Paul Graham in a discussion, and not only in writing an essay, and how would one go about structuring the inputs/outputs?

⌃ | ⌄ · Reply · Share ›

karpathy Mod ➦ oggy · an hour ago

yep this is very easy and I'm sure many people are looking into it already. (Not easy with this particular code, but in general easy)

⌃ | ⌄ · Reply · Share ›

LE · 6 hours ago

Amazing stuff, mr Karpathy! As a relatively fresh machine learning student and enthusiast, this was a really awesome tech demo. I wonder, do you use stochastic gradient descent to learn the parameters for your RNN? If so, how do you compute the gradient of the loss function, and which loss function did you use?

Interesting to see this Torch, perhaps that in itself might be reason to learn Lua -- currently, my language of choice is Python, and I'm happy to say that it seems like a lot of the ML community is on that track as well.

⌃ | ⌄ · Reply · Share ›

karpathy Mod ➦ LE · an hour ago

Yep, it's SGD with RMSProp adaptive learning rate. Loss function is cross-entropy (softmax classifier) for every next character

⌃ | ⌄ · Reply · Share ›

Mohan Radhakrishnan · 7 hours ago

After Andrew Ng's coursera Octave code this is the first time I come across code. Hope the git code is simple to understand. Thanks. Is there some way to visualize the NN ? I found http://bicorner.com/2015/05/13... and I am familiar with 'R" but the article is dense.

⌃ | ⌄ · Reply · Share ›

adv_rg · 16 hours ago

Do you think RNN's could do a better job in sentiment extraction than the existing machine learning models. Would be great if you could give some pointers. I work for a company which does this (http://www.reviewgist.com/htc-...) and looks like RNN's could give us a huge jump.

⌃ | ⌄ · Reply · Share ›

elderprice · 19 hours ago

Hi Andrej, are the character inputs "fixed" as one-hot vectors, or are they allowed to be fine tuned? What would happen if you randomly initialize character vectors and be fine tuned? What would happen if you randomly initialize character vectors and fine-tuned them?

I tried some sentence level classification tasks and found that performance was essentially same as random, unless I randomly initialized the character vectors and "fine-tuned" them during training.

^ | v • Reply • Share ›

karpathy Mod ➦ elderprice • 17 hours ago

Hello, In this particular case it doesn't make sense to finetune the one-hot input vectors because they are are acted on with a linear transformation in the LSTM, so the learned encoding vectors can be "absorbed" into those weights; i.e.: W_1(W_2 x) = (W_1 W_2) x. Kind of hard to explain in text :)
(in other words if you did this i'd expect about equal results. The learning dynamics would be a bit different but I don't have a clear intuition as how that would impact things)

^ | v • Reply • Share ›

Zzzz • a day ago

@karpathy

I remember Graves model was using one byte inputs, and it was generating unicode chars using "half character" at a time. Is this works similar way(so model could generate any char)?

Essentially I'm asking if I could feed model any file, or it works with limited alphabet/subset of characters?

^ | v • Reply • Share ›

karpathy Mod ➦ Zzzz • a day ago

this works the same way, you can feed in anything.

1 ^ | v • Reply • Share ›

Zzzz ➦ karpathy • 21 hours ago

It would be interesting to try to train such model on a file where every second sentence is a translation of a previous one to different language. I mean if we prime net at a test time with sentence in original language - will we get a translation? Probably a crappy one, but would be interesting to try! :)

^ | v • Reply • Share ›

Noah Barr • a day ago

@karpathy

Would the returned samples from PG/Shakespeare/Wikipedia examples be of higher quality if you used a word-level language model instead of character model with similar parameters?

I was curious if the overhead of learning how to spell words (vs a pure task of sentence construction with word objects) out weigh the reduction in sample set size?

^ | ⌄ • Reply • Share ›

Zzzz → Noah Barr • a day ago

If you use word vectors you will limit output of the model. So it would be hard to model URLs, code, spell proper names and forget about words in non latin alphabets(like ones you could sometimes see in English Wikipedia articles).

On the other hand there was a paper from Sutskever investigating mixed character-subword encoding, with promising results. But that model was more complicated.

^ | ⌄ • Reply • Share ›

mikeatlin → Zzzz • 21 hours ago

I was thinking that phoneme level encoding would be pretty cool. I bet it would get the meter right in shakespeare then!

^ | ⌄ • Reply • Share ›

Ken Bolinsky → mikeatlin • 8 hours ago

The dataset for accented syllabification would be ginormous! The wordsmithing necessary to generate iambic pentameter might be beyond the capability of an RNN...

^ | ⌄ • Reply • Share ›

mikeatlin → Ken Bolinsky • 7 hours ago

There's actually a pretty complete free machine-speech dictionary. It doesn't handle everything, and you sometimes need to add stuff, or write logic to switch between different pronunciation of homophones, but it's actually pretty reasonable. I used it when I was fooling around with a sonnet recognition script. It's pretty trivial to explicitly write something that recognizes iambic pentameter, and I'd BET a RNN could manage it...

...challenge accepted! When I have some free time =)

http://www.speech.cs.cmu.edu/c...

^ | ⌄ • Reply • Share ›

Kevin • a day ago

I've got a question regarding the structure of the model, if you have 512 hidden nodes per layer and take in 100 characters as input, are you working with a model structure similar to (4) in figure 1, with something like a delay of 412 nodes between the input and output?

^ | ⌄ • Reply • Share ›

karpathy Mod ➦ Kevin • a day ago

We're actually working with diagram (5) - the synced many-to-many case. We're training the RNN to output the immediately next character in sequence. So if a batch has 100 chars in the input length, then there are exactly 100 inputs and 100 outputs: the 100 outputs are shifted by one into the future, if that makes sense.

^ | ⌄ • Reply • Share ›

Kevin ➦ karpathy • a day ago

I see, yeah that's what I was thinking. However I'm confused as to where the 512 hidden nodes comes in.

^ | ⌄ • Reply • Share ›

karpathy Mod ➦ Kevin • a day ago

In my RNN figure in the seciton "character-level RNN", the hidden state size is 3 (the green layer): there are 3 numbers that make up the hidden state vector. This is the layer between inputs and outputs. So instead of 3 you'd have 512.

^ | ⌄ • Reply • Share ›

Kevin ➦ karpathy • a day ago

Oh ok. I was a little thrown when you wrote hidden nodes, and so I assumed that was the number of lstm/rnn units you wanted in the green layer. Thanks for the clarification.

^ | ⌄ • Reply • Share ›

George Hotz • a day ago

Can you share the command line flags(hyperparameters) you used for training Shakespeare? When I try just setting size=512 and layers=3 on the included data/tinyshakespeare, the network diverges.

Awesome stuff btw, I think this is a great step toward the ImageNet equivalent of RNNs, i.e. a network everyone knows about and can play with, except for language instead of images.

^ | ⌄ • Reply • Share ›

karpathy Mod ➦ George Hotz • a day ago

that's odd, i rarely ever see it blow up, it's usually very robust to different architectures. My exact parameters were:

```
{
print_every : 1
data_dir : "data/shakespeare"
seq_length : 50
batch_size : 100
num_layers : 3
gpuid : 0
checkpoint_dir : "cv"
decay_rate : 0.95
eval_val_every : 1000
savefile : "shake2"
seed : 123
learning_rate : 0.002
dropout : 0.25
grad_clip : 5
max_epochs : 50
rnn_size : 512
}
```

(and i'm using the full shakespeare dataset, not the tiny version that's included in the git repo, but the one that can be downloaded in the datasets link)

⌃ | ⌄ • Reply • Share ›

George Hotz ➦ karpathy • 18 hours ago

So I'm home from work now and trying this on my personal laptop. Works great!

There must be something broken with my work torch installation. Super weird way of presenting itself though, as numerical errors. Will follow up tomorrow, be nice to use a Titan X instead of the MBP internal.

⌃ | ⌄ • Reply • Share ›

George Hotz ➦ karpathy • a day ago

Hmm, that is odd. Grabbed whole dataset and tried, same issue. Tried with and without GPU

http://pastie.org/10201346

⌃ | ⌄ • Reply • Share ›

karpathy Mod ➦ George Hotz • an hour ago

do you have a recent and updated version of torch? The code for rmsprop changed.

⌃ | ⌄ • Reply • Share ›

David Sanders • a day ago

Very cool. This reminds me of some of Douglas Hofstadter's ideas about consciousness arising from some form of recursion or self-reference in the brain.

∧ | ∨ • Reply • Share ›

tborenst ➦ David Sanders • 13 hours ago

Could you point me to somewhere I could read more about this?

∧ | ∨ • Reply • Share ›

Andrew Clegg ➦ tborenst • 6 hours ago

http://en.wikipedia.org/wiki/G...

1 ∧ | ∨ • Reply • Share ›

Brent Lehman ➦ tborenst • 6 hours ago

Godel Escher Bach is all about this. He doesn't really get into it until the last half. Fascinating read though.

1 ∧ | ∨ • Reply • Share ›

Mark • a day ago

Thank you so much! Add it to your next class, I am sure your students will appreciate it. BTW, has anyone used them to analyse videos, say you are learning to ballroom dance and want feedback that a machine may capture but your instructor does not. Basically identifying movement behaviours.

∧ | ∨ • Reply • Share ›

JamesL • a day ago

Indeed, this is a very nice blog post. Thank you for sharing. Have you experimented with using an RLU transfer function layer and initializing the recurrent weight Whh at the identity (or epsilon times the identity)? The recent paper by Le, Jaitly, and Hinton, http://arxiv.org/pdf/1504.0094..., provides some evidence that this technique (the authors call a model trained this way, an IRNN) leads to better performance for long-term dependencies. It would be interesting to see how the IRNN performs on the datasets you have used above.

∧ | ∨ • Reply • Share ›

karpathy Mod ➦ JamesL • a day ago

I haven't tried this. I also haven't heard of someone replicating the IRNN results yet.

∧ | ∨ • Reply • Share ›

JamesL ➦ karpathy • 8 hours ago

Thanks for reply.

∧ | ∨ • Reply • Share ›

Søren Kaae Sønderby • a day ago

very nice blog post. I have one question: Do you sample the previous prediction and use that as addtional input when you train? Do you know if that would be beneficial and when to use it?

⌃ | ⌄ • Reply • Share ›

karpathy Mod ➦ Søren Kaae Sønderby • a day ago

Thanks! Good question;

Short answer: I don't

Long answer: There are two modes of operation: either you feed in the next ground truth character or you feed in a sample from the model. By default people tend to use the first way (this code does as well), but it could be argued that the second way could make the network more likely to recover from its own errors. I don't have too much experience with the second way - I tried it once and it worked worse, but it makes sense to me that a clever blend of both strategies could generalize better.

1 ⌃ | ⌄ • Reply • Share ›

Søren Kaae Sønderby ➦ karpathy • a day ago

I did play with is aswell and I never got i to work any better. Abut libraries, I'm working on LSTM/GRU/RNN support in lasagne (https://github.com/Lasagne/Las.... It'll hopefully be merged within a week. (Eventhough you don't like theano)

2 ⌃ | ⌄ • Reply • Share ›

Ilya Flyamer • 10 hours ago

Have you tried feeding in the human genome? It's not very straightforward to analyse the result though...

⌃ | ⌄ • Reply • Share ›

ALSO ON **ANDREJ'S BLOG** WHAT'S THIS?

Lessons learned from manually classifying CIFAR-10

3 comments • a year ago

karpathy — fixed.

Interview with Data Science Weekly on Neural Nets and ConvNetJS

1 comment • a year ago

Guest — I am a student from China. The interview ariticle really helped me a lot. Thank you very much Karpathy. You are

Quantifying Productivity

6 comments • 10 months ago

karpathy — Are you kidding? Weekends are the best time to get work done - emptier offices, no meetings, no

Feature Learning Escapades

6 comments • a year ago

Phong — An interesting discourse. I have some thoughts on your argument of unsupervised learning: 1. Let's see how

Andrej Karpathy blog

karpathy
karpathy

Musings of a Computer Scientist.